Project Number: MLC-RL98

Development of a CLARiiON Fibre Channel Device Driver for Linux

A Major Qualifying Project

submitted to the Faculty

of

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Charles Anderson

Frank Sweetser

May 4, 1999

Approved:

Professor Mark Claypool

1. Linux
2. Fibre Channel
3. RAID

Abstract

Modern applications, such as high-end databases and multimedia applications, are placing higher bandwidth demands on storage devices. Currently, the best choice for both performance and reliability is a Redundant Array of Inexpensive Disks (RAID). CLARiiON, one of the leading developers of high-end RAID devices, is seeking to deploy Linux on their servers as an alternative to DG/UX, their in-house version of UNIX. In this project we have begun to implement a Fibre Channel RAID driver for Linux.

Contents

List of Figures

Chapter 1

Introduction

In this age of cheap, fast CPUs with large on-chip caches and large amounts of RAM, I/O bandwidth remains one of the most limiting factors in the performance of most computer systems. Also, hard drives are probably the most unreliable part of any computer system. Due to their ever-increasing speeds and capacities and shrinking sizes, the mechanical parts of hard drives are even more prone to wear and eventual failure than in the past. These two factors have contributed to a vast amount of effort by researchers and industry in order to find solutions to the problems of performance and reliability of I/O subsystems.

Several solutions exist today to address the performance and reliability of I/O subsystems. In general, there are two parts of an I/O subsystem that must be considered: the storage devices, and the data channel between the CPU and the storage devices. For lower end servers, SCSI is the most popular choice for the data channel, with several independent SCSI disks attached to provide the data storage. This solution performs fairly well, and has the advantages of being fairly flexible and cheap. The disadvantages of this approach, however, are twofold. First, SCSI is limited in distance and speed, and may not be adequate for higher-end server requirements. Second, the issue of reliability remains to be solved. To address the first issue, Fibre Channel was developed as an alternative to SCSI. To address the second issue, Redundant Array of Inexpensive Disks, or RAID, was developed to provide fault tolerance.

1.1 CLARiiON

CLARiiON Advanced Storage Solutions, a division of Data General Corporation, develops high-end RAID storage and end-to-end Fibre Channel solutions.[1] Many of their current storage solutions are based on their AViiON line of servers running Data General UNIX (DG/UX), which is no longer being developed. CLARiiON sponsored this project with WPI to develop a Fibre Channel driver for Linux, to be used as a possible replacement for DG/UX on their AViiON servers. To this end, CLARiiON lent WPI a 10-slot Fibre Channel RAID, an AViiON server, and two Emulex LP6000 Fibre Channel PCI cards for this project. We have used that equipment, with Linux running on the AViiON server, to develop a device driver for the Emulex LP6000 card.

1.2 Motivation

There were two primary motivations for this project, both on our part and CLARiiON's, to write this device driver. The first was to replace DG/UX, Data General's in-house version of UNIX. CLARiiON currently uses DG/UX as a development platform for their RAID development. They also ship complete storage solutions based on DG/UX, among other platforms. Over the years, it has become too expensive for Data General to continue maintaining DG/UX—CLARiiON's business is RAID, not UNIX. By using Linux, CLARiiON gains many advantages. Linux is freely available at no licensing charges, even for a full source license. It is available to anyone including CLARiiON and all its customers, allowing CLARiiON to subtract the cost of DG/UX from the price tag. Linux is being constantly developed, at no cost to CLARiiON. In addition, the customer will have a wide range of Linux distributions to choose from, any one of which will work with exactly the same kernel and LP6000 drivers.

The other reason for developing this device driver was to contribute work towards a Fibre Channel layer for Linux. At the beginning of this project, there was very little work that had been done in the mainstream kernel for true Fibre Channel support. Our goal was to provide a full featured subsystem, including support for multiple cards and Upper Layer Protocols, and abstracting as much as possible that is common Fibre Channel functionality, such as the login procedure. While our project only got as far as designing the basic layout, others in the Linux community have begun and made large amounts of progress in this area independently. Their work has already begun appearing in the official kernel tree.

1.3 Technologies

There are three core technologies that we have directly dealt with: the Linux operating system, Fibre Channel, and RAID. Linux is a clone of Unix written by Linus Torvalds beginning in 1991, building off the GNU project. Fibre Channel is a high speed interconnecting technology, used primarily for very high speed disks and direct connections between computers. RAID covers a number of ways of combining several physical disks and presenting them as a single logical disk for the purpose of gaining speed, fault tolerance, or varying combinations of the two. By running SCSI over Fibre Channel, a combination of very high speed, reliability, and flexibility can be achieved.

1.3.1 The Linux Operating System

Linux[2] is a freely available, freely redistributable UNIX-clone operating system. The kernel was developed by Linus Torvalds while he was a Computer Science student at the University of Helsinki, Finland, while the other core components of the operating system, including system libraries, compiler, and other system utilities were taken from the work of the Free Software Foundation[3](FSF). The Linux kernel, and all software released by the FSF, is released under the GNU General Public License[4](GPL).

While most licenses restrict the redistribution or modification of software, the GPL instead ensures that anyone who obtains a copy of GPL-licensed software has the right to freely modify and redistribute the software as much as they like, with two restrictions. First, any redistributions must also be released under the GPL. Second, you must release all source code modifications you have made, thereby giving the next person down the line an equal chance to view and modify the software. As a result, hundreds of people have read, debugged, and enhanced the popular GPL software packages, making them typically more stable and full-featured than their commercial counterparts.

For CLARiiON, the social and moral reasons for releasing and using software developed under the GPL are secondary. Rather, the fact that Linux is a stable, efficient, rapidly developing operating system means that they will be able to deliver products to their customers for less total cost than maintaining DG/UX, the in-house UNIX they currently ship on all servers. The liberal redistribution policy of Linux also makes it an ideal platform for projects such as ours. The full source code can be easily obtained with few worries about restrictive licensing, red tape, or fees. And, best of all, all work done by this project may be freely published without getting further agreement from any of the authors, as this is explicitly allowed by the GPL.

1.3.2 Fibre Channel

Fibre Channel[10] is an I/O channel and network technology that was developed and standardized by ANSI in 1994 to provide a high-performance, flexible architecture to interconnect storage systems, computers, and networks together over the same physical medium. Fibre Channel combines the best aspects of both networking and I/O channel technologies, and provides compatibility by allowing existing network (IP, IEEE 802) and I/O (SCSI, IPI) protocols to run over it. There are several physical layer mediums specified by the FC-PH standard, utilizing copper and fiber optic cables at speeds between 100 megabits/sec (Mbps) and 4 gigabits/sec (Gbps), at distances up to 10 km. In addition, Fibre Channel provides for flexibility by defining point-to-point, arbitrated loop, and switched fabric topologies. Together these features make it an ideal solution for high-performance, high-reliability applications.

1.3.3 RAID

Sooner or later all hardware fails—it is simply a question of when it happens, and how prepared you are to deal with it. Most disk I/O systems, such as IDE and SCSI, are very poorly prepared to deal with hardware failures. To get around this, redundant configurations can be set up, the most common being RAID. There are a number of relatively inexpensive RAID solutions available, ranging from purely software based RAID with inexpensive IDE drives, to more expensive high performance SCSI hardware RAID controllers. Fibre Channel, while the most expensive, offers the greatest potential for both performance and flexibility.

Chapter 2

Background

Before we delve into the details of our project, we present a little background information on the core technologies that we have dealt with. In this chapter, we will discuss RAID, SCSI, Fibre Channel, and programming the Linux kernel.

2.1 RAID

The basic idea of RAID, which was developed in 1987 by Patterson, Gibson and Katz at the University of California Berkeley[7], is to combine multiple inexpensive disk drives into an array in order to increase the performance and/or reliability of the storage device. The different methods of combining the individual drives are called RAID *levels*. Patterson[7] originally defined RAID levels 1 through 5, described below. Since that, other configurations have been defined by various vendors. A popular addition is RAID-0, also described below.

2.1.1 Mirroring and Striping

Before discussing the specific RAID levels in detail, it is helpful to understand the basic building blocks upon which the RAID levels are built. The basic idea of RAID is to combine multiple physical disks into one logical or virtual disk. This is done by *mirroring* and *striping*.

Mirroring is just what it sounds like—maintaining several copies of the data, with one copy per physical disk. Mirroring increases the overall reliability of the data storage system, since if one disk fails, there are other copies of the data to recover from.

With striping, the data is split into fixed sized chunks that are distributed among all of the disks. Striping increases performance, since the I/O is split across multiple separate devices, but it decreases reliability, since if one disk fails, all the data is lost.

Each RAID level uses various combinations of these two techniques to increase both the fault tolerance and performance of the array.

2.1.2 RAID-0

RAID-0 is just simple striping as described above. The size of each stripe will affect the performance of the array. Typically, large stripes are used for I/O intensive applications, such as Usenet news servers. Smaller stripes (such as sector-sized stripes) may be used, but in that case the spindles of the individual disk drives usually must be synchronized to maintain adequate performance. Since no redundant storage of data is done in RAID-0, a loss of one drive will cause all data on the array to be lost. The capacity of a RAID-0 array is equal to the sum of the capacities of each drive in the array.

2.1.3 RAID-1

RAID-1 is plain mirroring. The read performance in this case will be n times that of a single disk, where n is the number of drives in the RAID-1 array. Since the data needs to be written to all disks simultaneously, the write performance remains unchanged. Reliability of a RAID-1 array is also n times that of a single disk, since there are always n copies of the data. The capacity of a RAID-1 array, however, is only as much as a single disk.

2.1.4 RAID-1/0

RAID-1/0 combines striping and mirroring together. This yields the performance of a pure stripe, along with the reliability of mirroring. This is the best case for both performance and reliability. It is also the most expensive, since it only gives the user n/2 disk space. For high availability situations, where there is typically not a large amount of data, it is a cost effective solution.

2.1.5 RAID-2

RAID-2 is similar to RAID-0 with sector-sized stripes, except some of the disks are used to store ECC (Error Correcting Code) information. This level is rarely used anymore due to most modern drives having built in ECC.

2.1.6 RAID-3

RAID-3, like RAID-2, uses striping with sector-sized stripes and drive spindle synchronization between the disks of the array. However, instead of storing ECC information, one drive is used to store parity information. This parity information can be used to recover the data on a single failed disk by exclusive-ORing it to the data on the other disks. Since each write operation requires reads to every disk in the array in order to calculate the parity information, write performance is not much improved over single disks. The capacity of a RAID-3 array is n-1 times the capacity of each disk in the array.

RAID-3 provides the best read throughput of all RAID levels while still providing some degree of protection against lost data. The best use for RAID-3 is reading large contiguous disk blocks, where RAID-3 avoids the drive head seek bottleneck. Small writes are not good for RAID-3, since the parity disk must be written to at the same time as the data disks. This makes RAID-3 more suited to high throughput tasks such as multimedia streaming.

2.1.7 RAID-4

RAID-4 is identical to RAID-3, except large stripe sizes are used to increase performance. Due to the larger stripes, an entire read operation will usually be from one disk, allowing multiple I/O operations to be overlapped. Writes, however, are not overlapped since the parity disk still needs to be updated. RAID-4 has been pretty much replaced by RAID-5.

2.1.8 RAID-5

RAID-5 removes the parity disk bottleneck of RAID-4 and RAID-3. Instead of storing parity information on a separate disk, RAID-5 rotates the parity among all the data disks. In this way, multiple reads and writes can be overlapped, since each operation will at most access two drives—the drive holding the data to be read or written, and the drive holding the corresponding parity information. Performance of RAID-5 arrays is not quite as good as RAID-1; however, they provide much better storage capacity for the same number of disks, as well as much better fault tolerance.

2.2 SCSI

SCSI, the Small Computer System Interface, is an I/O channel designed for connecting up to eight (and in later versions, sixteen) devices to a computer in a bus configuration. The original SCSI specification (now called SCSI-1), specified a synchronous 8-bit parallel interface running at 5 MHz, for a total throughput of 5 megabytes/sec. This was extended in several ways by increasing the bus speed, utilizing asynchronous transfers, and widening the bus, such that the latest SCSI systems are able to achieve a throughput of up to 80 megabytes/sec. All SCSI systems speak basically the same protocol, with the later versions adding various extensions. This protocol can also be used over other physical interfaces, such as Fibre Channel.

2.3 Fibre Channel

Fibre Channel is popular as a simple replacement for the highly complicated proprietary disk channels and storage area networks currently used in high-end computing environments. Fibre Channel uses several new concepts in the area of disk interconnections. Flexibility of configuration, cable topology, administration simplification, and cable length are all improved over SCSI.

The Fibre Channel specifications are structured into five independent layers that describe various levels of functionality. These layers are Physical Interface and Media (FC-0), Transmission Protocol (FC-1), Framing and Signaling Protocol (FC-2), Common Services (FC-3), and Upper Layer Protocol Mapping (FC-4). Below, we discuss the lowest three layers together as the Physical Layer, since more detail at those layers is not in the scope of this paper.

2.3.1 Physical Layer

As SCSI speeds have increased over the years with the introduction of technologies such as Ultra Wide SCSI, cabling and termination issues have become more and more critical. By using large numbers of parallel connections in a ribbon cable (often over 50 physical connections), crosstalk becomes a severe problem, demanding that nearly half of the connections are grounds. While SCSI may support as many as 15 devices on a single physical set of cabling, this is often difficult due to the distance limitations of the cable. This limitation is somewhat helped by requiring termination on each end of the chain, but this in turn introduces another element that must be carefully configured as well as another point of failure.

Fibre Channel is considerably more flexible. The physical layer specifications FC-0, FC-1, and FC-2 are defined by the FC-PH document. In a typical configuration, a 4 wire serial connection is used. Fibre Channel can run at speeds from 133 Mbaud to 4.25 Gbaud. Typical configurations run at 1.06 Gbaud. When the overhead due to the 8B/10B encoding is figured in, this has a theoretical bandwidth of about 100 MB/s. This is considerably better than the 40 MB/s that Ultra Wide SCSI can do. Also, future increases in speed up to 4.25 Gbaud are being developed, with a total throughput of about 400 MB/s.[10]

Copper Media

The more popular way to wire Fibre Channel devices is with either twin-axial copper cable or specially made twisted pair serial cables. For short lengths, such as the connections between disks, copper cable is much easier to use. Copper cable is more common, due to fiber optics being significantly more expensive and more sensitive to physical damage. Typical configurations with twin-axial cable can support distances of up to 100 meters. Television coaxial cable is another supported copper physical medium.

Optical Media

Fibre Channel can use an optical carrier too, thus making campus wide networks, remote replication, and storage area networks a reality for data centers and distributed computing sites. The reader should not infer that Fibre Channel means fibre optic carrier; Fibre Channel is more or less independent of the physical layer. Singlemode and multimode fiber optics are supported at distances of several hundred meters to over 2 kilometers.

2.3.2 Topologies

Fibre Channel disks and arrays can be connected by point to point, arbitrated loop, or switched fabric type connections. This allows for more advanced configurations, such as multiple servers connected to a single disk. While such a configuration is very difficult with SCSI, it is routine with Fibre Channel.

Point-to-Point

Fibre Channel ports can be connected to each other in a point-to-point configuration, either in a manner similar to a crossover cable used for ordinary RS-232 serial ports, or via a switch which sets up the point-to-point connection. In a point-to-point configuration, the two devices have a dedicated bandwidth, full-duplex connection to each other.

Arbitrated Loop

Arbitrated Loop is a shared bandwidth connection that allows up to 127 devices to be connected in a daisy-chain fashion. Each device arbitrates for access to the loop, and then sets up a logical connection to the target device. Only one connection on the loop can be open at any one time. During this connection, the source and target devices have full access to the total bandwidth available on the loop. Arbitrated Loop is the preferred connection topology for connecting disk arrays; it is designed as a direct replacement for SCSI.

To alleviate the problems with broken loops when a device in the loop is powered off or removed, a Fibre Channel hub can be used to create a physical star topology. In this case, if the loop is broken, the hub will complete the loop, bypassing the disconnected or powered off device. This is similar in operation to token ring concentrators.

Switch Fabric

If an arbitrated loop does not provide enough bandwidth for the devices using it, it is also possible to use a switch to increase capacity. This provides what is physically a star topology, as well as many simultaneous virtual point to point connections. A switch will also allow a device to disconnect from a loop without taking down every other device. A Fibre Channel switch can provide circuit switched service, frame switched service, or both. In addition, a switch can be used to interconnect individual devices (N_Ports), arbitrated loops (NL_Ports), and other switches (E_Ports). When many switches are connected together in this manner, the entire collection is referred to as a *fabric*. This flexible functionality is configured completely automatically upon startup of the switches, and allows for great scalability of the Fibre Channel system.

2.3.3 Classes of Service

Fibre Channel FC-2 provides several classes of service for connections made between two devices. These classes are described below:

Class 1 - Acknowledged connection-oriented service

This class of service is a dedicated bandwidth connection with guaranteed quality of service (QoS). The connection must be set up with a handshake sequence before it can be used. Each frame is delivered in-order and receipt is acknowledged. Once the connection is no longer needed, it must be disconnected with another handshake sequence. While two devices have a Class 1 connection open, no other device can talk to them (except when Intermix mode is used—see below).

Class 2 - Acknowledged connectionless service

This class of service is similar to Class 1, except no connection set up sequence is needed. A device simply transmits Class 2 frames whenever it wants, and receipt of the frames is acknowledged by the destination. Each frame may follow a different path in the switch fabric, since no path is dedicated for this class of service.

Class 3 - Unacknowledged connectionless service

Class 3 is just like Class 2, except no acknowledgement of frames is done.

Class 4 - Fractional bandwidth connection-oriented service

Class 4 is a connection-oriented protocol similar to Class 1. In the case of Class 4, however, only a fraction of the total available bandwidth is dedicated for the connection. Quality of service parameters are used to set up the desired bandwidth characteristics for each direction of the connection independently. Delivery is guaranteed in-order with acknowledgements.

Class 6 - Simplex connection service

Class 6 is similar to Class 1 in simplex mode, providing unidirectional and multicast connections. This is ideal for sending the same data to many devices at the same time.

Intermix

Intermix is a special mode of Class 1 connections. In this mode, the full Class 1 bandwidth is dedicated for the connection, but other Class 2 and Class 3 connectionless traffic can be sent to the two devices when the bandwidth is not otherwise being used for the Class 1 connection.

2.3.4 Upper Layer Protocols

Mapping Upper Layer Protocols (ULP's) to Fibre Channel is defined in the FC-4 documents. Several protocols are supported simultaneously over a single Fibre Channel system. Each protocol is defined in its own FC-4 document. A few of the more important protocols for which FC-4 mapping specifications have been written include SCSI (FCP), Intelligent Peripheral Interface (IPI), Internet Protocol (IP), and IEEE 802 (FC-LE).

2.4 Programming the Linux Kernel

While nearly all of the Linux Kernel is written in C, including all of our work, there are still a few difficulties not present in a typical program. The first of these is that the standard C library functions, such as string handling, network and file I/O, and advanced memory management are not available. In some cases, such as string handling, some of the simpler functions have been re-implemented, though they are much simpler than the more advanced versions found in a full blown C library. Other ones, such as simple memory management, have Application Programmer Interfaces (API's) very closely related to the standard C version (i.e., `kmalloc()` instead of `malloc()`), and still others, such as network and file IO, are sufficiently different and difficult to handle directly that they are rarely if ever used directly.

A kernel programmer must also be more aware of the resources used, as the limitations placed upon user space programs do not apply to the kernel. Any memory allocated by the kernel must be held in physical RAM at all times, meaning that it cannot ever be swapped out. Any kernel code can disable interrupts if it so desires, or spin in an infinite busy loop, effectively locking up the entire system. And, with full access to any and all system and device I/O ranges, the driver must be careful only to access resources known to be safe.

Luckily, there are standard API's for nearly all of these problems. There are routines to query if resources have been claimed by any other drivers, ways to schedule large portions of work without locking up the entire system, and enough information is readily available directly from the kernel such that, beyond data buffers, it should rarely be necessary to allocate large chunks of memory. As long as the API's are properly used and a little thought and care is put into the driver, none of these are overly large problems.

2.4.1 IRQ Handling

Setting up interrupt handling under Linux is fairly straightforward. Once the IRQ used by the device is determined[1], all that must be done is calling the `request_irq` function and passing it the IRQ number to be used and the function to be called whenever that IRQ is raised. From that point on, the kernel will call the handler function whenever it detects that IRQ. The kernel will also prevent any other device drivers from registering that IRQ, unless both drivers have been written to allow IRQ sharing. Note that the hardware must also support IRQ sharing; it is assumed that the device driver will have knowledge of this and pass the appropriate flags to `request_irq`. Likewise, the `free_irq` function may be called to release the IRQ handler. This can be useful for loadable modules, or for devices that will only use the IRQ when the device is opened and in use, for example.

While the IRQ handler is running, interrupts should be disabled. Since no other interrupts can be detected during this time, it is in the best interest of overall system stability and performance for the interrupt handler to finish and re-enable interrupts as soon as possible. Since there often is lengthy work to be done, such as computing checksums on packets or copying large amounts of data into and out of buffers, the *bottom half handlers* are

[1] This can be difficult with some devices, particularly ISA ones where autoprobing can be dangerous - luckily the LP6000 card is a PCI device, which greatly simplifies matters.

frequently used.

2.4.2 Bottom Half Handlers

Often times, there are situations where a kernel author wants to defer a block of work until later. The most common situation is in the interrupt handlers, where the actual handler wants to do its work as quickly as possible and exit so that interrupts may be re-enabled. To meet this need, the bottom half handler code was created.

To register an instance of a bottom half handler, the `queue_task()` function must be used. The three most important pieces of information passed are the function to be called, a single argument passed to the function (typically a pointer to a struct containing more information), and which bottom half queue is to be used. The function `mark_bh` must then be called on the queue to which the function was just added.

Exactly when the function gets called depends on which queue was used. The maximum number of bottom half handlers is currently a fixed number, 32. Since adding new bottom half queues requires modifying core kernel code directly, and there is no API to do so, it is very rare for device drivers (which are often distributed as a stand alone source code file separate from the main kernel) to create their own queues. There are several task specific queues that may be used, including one for network activity, and one for console I/O activity. The most commonly used generic queue is `IMMEDIATE_BH`. This queue runs immediately on return from interrupt handlers and system calls.

2.4.3 PCI Handling

One of the greatest advantages of PCI devices is intelligent autoconfiguration. With older ISA devices, the user must manually configure each device, ensuring that there are no resource conflicts. With PCI devices, the PCI BIOS ensures that all PCI devices are configured properly at each bootup. This allows for virtually trouble free automatic configuration, including easy handling of added and removed devices with no configuration changes required by the user. In addition, the fact that all configuration is done by the PCI BIOS allows the operating system to easily obtain all configuration values by simply querying the PCI BIOS itself, rather than having to probe all present PCI devices separately. Once the generic resources, vendor ID, and card ID in use by each card is known, then further device specific probing and configuration may safely be done without danger of interfering with an unknown device.

The Linux kernel currently provides two PCI API's. The first one, the `pcibios_` class of functions, is provided for backwards compatibility. It provides a function to find a PCI device based upon the vendor and/or card ID, and a pair of functions to read and write to the PCI configuration space of that particular card. Device specific information is kept in a `struct pci_dev`. Values of offsets to commonly used information, such as I/O regions and IRQ's, are provided as `#defined` values.

The second API, the `pci_` class, is much newer. It is only present in the development kernels 2.1.90 and newer. It is much simpler to use, however.

The primary difference is that much of the information which previously could only be obtained by accessing the PCI configuration space for a particular card is now available in the `struct pci_dev` for that device. Also new is the `pci_set_master` function. With the previous API, to configure a card for bus mastering operation, the author would simply rewrite the PCI command word. With newer kernels, this step is much easier—one simply calls the `pci_set_master` function. In addition to writing out the appropriate word, this function will adjust PCI latency timings, and perform any other kernel and PCI BIOS configurations needed for proper bus mastering operation. More documentation on the PCI API, in particular the `pcibios_` one, can be found in the Linux kernel source tree under `Documentation/pci.txt` and `include/linux/pci.h`.

2.4.4 Memory Management

Basic memory management is, from the point of view of a device driver, fairly simple. For most small to medium sized memory requirements, `kmalloc` and `kfree` can typically be used. The `kmalloc` function is almost identical to the standard C library `malloc` function, the biggest difference being that it takes two arguments instead of just one. The first argument for `kmalloc`, like `malloc`, is the amount of memory in bytes. The second argument is the priority. Two priorities are typically used, either `GFP_KERNEL` or `GFP_ATOMIC`. The difference between the two is that `GFP_ATOMIC` is allowed access to a special reserved pool of memory that is set aside for situations where waiting to swap out a page is not an option, such as during an interrupt. When `GFP_KERNEL` is used, on the other hand, the calling routing must be prepared to block while the system attempts to free up the needed memory.

When larger amounts of memory are needed, however, it is more efficient to use a page oriented scheme. There are two sets of functions for doing this. The first set is used for grabbing pages directly. They are `get_free_page`, which returns a pointer to a new page, `__get_free_pages`, which returns a pointer to a multiple page-sized contiguous memory area, and `__get_dma_pages`, which returns a pointer to a region suitably located for ISA DMA. In order to free memory allocated by these functions, either `free_page` or `free_pages` should be called. All of the allocation functions take a priority, with semantics identical to the priority passed to `kmalloc`. The multiple page versions (`__get_free_pages` and `__get_dma_pages`) also take an order argument. An order of n indicates that 2^n pages should be allocated.

The other method that may be used is the `vmalloc` function. This function takes one argument, the size of the memory block requested. It works by requesting enough pages to make up the needed amount individually with `get_free_page`, and then modifying the page tables to make the newly requested pages appear to be contiguous in memory. In reality, though, the pages may be highly fragmented across memory, and the addresses returned are always beyond the actual highest physical memory address. This fragmentation makes `vmalloc` unsuitable for allocating buffers to be passed to devices, as they nearly always expect the buffer to be contiguous. In order to free the memory, the `vfree` function should be used.

In certain extreme cases, even more contiguous memory than this may be needed. In particular, note that `get_free_pages` will only take a maximum order of 5, for a maximum of 32 consecutive pages. The only method left then is by grabbing the memory at boot time. During the boot sequence, each driver has a chance to modify the upper bound of memory. The driver can lie about the new upper bound by lowering it to give itself enough room to work in. Since the kernel no longer believes that region of memory exists, the driver is free to work with that range as it will. Note that this memory can never be returned to the kernel, and this method will only work for compiled in drivers, not for loadable modules.

2.4.5 Accessing I/O Addresses

Typically, PCI devices will have one or more sections of memory to be mapped out for the purpose of transferring data to and from the device. The PCI BIOS determines the exact address this memory is to be mapped out to, typically in a BIOS dependent range near the top of the address space. On some architectures, this may be mapped into the main memory address space; on others, the I/O buses may have their own separate address space.

In either case, this memory needs to be made accessible before it can be used. At boot time, the page tables are only set up with an entry for each page of physical or virtual memory. Therefore, the `ioremap(address, size)` function must be called. Very simply, it adds the needed entries into the page tables for the mapped I/O range. If this is not done before the memory is accessed, then the Memory Management Unit (MMU) will indicate a request for an invalid memory address, causing an invalid page fault and kernel panic.

Once the memory has been remapped with `ioremap`, a bit of care must still be taken accessing it. On some architectures, the address returned by `ioremap` is not directly usable by the CPU. The driver cannot directly dereference the address, but rather must go through the right address space. Therefore, to be portable, the `readX` and `writeX` functions (X equals b for 8 bit byte, w for 16 bit word, and l for 32 bit long) must be used for all access to the I/O addresses.

2.4.6 The SCSI Layer

The only portion of the Linux SCSI layer which we concerned ourselves with is that of writing a low-level SCSI driver. The bulk of the SCSI logic is contained in the middle and upper SCSI layers. When a SCSI driver is initialized at boot time, or loaded later during run time, it registers itself with the upper layers by passing a struct containing various flags indicating card and driver properties and abilities, and several functions. There are four key sections that the driver presents to the upper layers: initialization, the SCSI command handler, the interrupt handler, and the abort function.

The initialization code is called during boot time or module load time. It is expected to find all cards handled by that driver, configure them into a known ready state including reserving IRQ and I/O ranges, and register each one with the upper SCSI layers.

The SCSI command handler takes SCSI commands from the upper layers that have been issued as a result of events such as disk reads and writes. The typical method is for the device driver to add the SCSI command to a driver-specific queue for processing later. Our driver uses a bottom half handler to defer processing of the command to a later time. In addition to the SCSI command itself, a pointer to a function is also passed. Once the driver has finished processing the given SCSI command, this function must be called on the command to signal to the upper SCSI layers that this command has been fully processed. If this function is not called within a given timeout period, then the upper layers assume that an error has occurred and attempt to abort the command.

The interrupt handler is invoked whenever the card raises an interrupt. It must determine which section of the card raised the interrupt, copy any pertinent information from the card into host memory for later processing, and clear the relevant interrupt bit on the card. Like the SCSI command handler, the interrupt handler defers the bulk of the processing work via a bottom half handler.

The abort function should only be called when an unrecoverable error has occurred. It is called upon a particular outstanding SCSI command when the error is detected. For example, if the upper layers detect a timeout, they will call this function. The abort function is expected to dequeue the command entirely.

Chapter 3

Goals

For this project, we have 3 primary goals:

1. Begin implementing a Linux driver for the Emulex Light Pulse 6000 Fibre Channel chipset.

2. Begin implementing Fibre Channel under Linux for x86.

3. Perform and document stability and performance testing.

3.1 Code Linux driver for the Emulex card

When we originally began writing the driver, we were working with the 2.1 series of the Linux kernel, considered the developmental branch. Recently, this branch has been upgraded to stable status, and re-released under version 2.2. At the time of this writing, this branch is up to release 2.2.7. Initial reports of stability, performance, and a wide range of new device drivers are very promising for the future of the 2.2 branch.

Like the 2.1 kernel, the 2.2 kernel includes a number of features important to mid- to high-end servers. These features include the ability to use a large amount (over 1Gig) of RAM, the first stages of fine grained SMP kernel locking as opposed to the single kernel lock in the 2.0 kernels, a much more advanced filesystem caching scheme, and greatly improved support for other architectures, including SPARCStations, UltraSPARCS, and Alphas. The last feature in particular has the greatest potential for widespread use of the driver. With a little care given to portability issues such as host endianess and data structure sizes, the exact same driver can drive the Fibre Channel card on any machine with a PCI bus, from a midrange Pentium class machine, all the way up to powerful UltraSPARC or Alpha Server. PCI dictates the byte ordering of I/O (all PCI cards are little endian), so tracking byte order will not be problematic. Additionally, the LP6000 card is compatible with the Open Boot PROM found in Sun Workstations.

The biggest difference between v2.0 and v2.2 with respect to SMP performance is the use of spinlocks instead of globally disabling interrupts. Under v2.0, kernel data structures were protected by simply masking interrupts on all but the one processor entering kernel mode. This meant that only a single CPU could be active in kernel mode, but it also severely hampered scalability, as any other processors waiting to run kernel mode code would be sitting idle. In addition, this meant that incoming interrupts could only be serviced on a single CPU. In v2.1, spinlocks were introduced. Spinlocks are essentially a processor oriented mutex, one per critical code section, with full read/write and read only locking ability. While the the locks are still quite coarse (i.e., very large sections of code, such as the filesystem or network code, are protected by a single spinlock), this sacrifices enhanced scalability for simpler code, meaning less likelihood for damaging bugs. Over time, the locking will be progressively fine-tuned (most likely in the v2.3 development tree, currently yet to be released), giving progressively enhanced scalability while minimizing the rate that bugs are introduced.

3.2 Implement Fibre Channel under Linux

One of our goals was to implement a full Fibre Channel layer for Linux. While we were not able to begin actual work on such a layer, we were able to design what such a layer might look like. At the beginning of this project, there was virtually no formal support for a true, multi-ULP FC layer under Linux. The v2.2 kernel includes an initial version of just such a layer. While it is still in its earliest stages, any future work projects towards Linux Fibre Channel would almost certainly be better spent working on this layer, which is already considered a part of the official kernel proper, than beginning anew.

In practice, Fibre Channel is often treated as a special case of a SCSI host adapter. For our purposes, we have only concerned ourselves with the SCSI protocol over Fibre Channel. Our driver was developed as a SCSI driver as far as the Linux kernel was concerned; therefore, other ULP's, such as IP, were outside the scope of this project.

3.3 Stability and Performance Testing

As would be expected with such a project, we had planned to perform testing to ensure that the driver and whatever Fibre Channel layer work we performed would be stable enough and deliver acceptable performance for real world use. However, since we were unable to complete a fully functional driver, this goal must be passed on to the next group to continue the project.

Chapter 4

Methodology

This chapter details how we researched the project, how we did our development, what resources were used, what difficulties we encountered, and how we attempted to overcome them.

4.1 Linux Emulex Driver

On the Linux side, all source code is freely available, and there are several public discussion forums where Linux development issues may be discussed, often with the primary developers. There is also a good deal of existing documentation on writing various device drivers for the Linux kernel, including vast amounts of online[11] and published[5] documentation. We utilized these resources to help in our development of the driver.

As for the Emulex card, while the documentation is a little harder to find, CLARiiON was able to provide us with a complete set of documentation. While the documentation did not appear to be the final released version, but rather a prerelease draft, it was invaluable in the project. The Emulex card we used, the LP6000, is a fairly intelligent card, which simplifies the device driver. While simpler cards need large amounts of polling and hand-holding, with large amounts of functionality required in the driver, intelligent cards such as the LP6000 can do the bulk of the work themselves.

When driving a simple card, the driver is required to help the connected devices through many smaller steps, requiring the driver to do large amounts of busy-work. This ends up using unacceptably large amounts of CPU time for a server, and also increases driver complexity and potential bugs. With a more intelligent card like the Emulex card, the driver simply assembles a data structure describing the request, and passes it to the card. The card returns a request ID, and later raises an interrupt when the request has been fulfilled. This moves the burden of cost and complexity into the hardware level, away from the operating system.

With the large amount of buffering that is done by the card, an aggressive interrupt strategy may not be the best solution[7]. The overhead added by frequent interrupts may not be justified, since there is no pressing need to clear data from buffers, and it is likely that this will not change performance. Consuming too many interrupts can be a bottleneck to scalability, since an interrupt will cause an involuntary context switch, something that is very expensive. As throughput grows, handling interrupts takes more and more time, and fewer resources are available for doing actual work. Eventually, polling can even become more efficient. Of course, if polling is more expensive than an interrupt, nothing is gained by using it.

In the end, we were unable to achieve a fully functional driver for the card. The greatest single difficulty was that either due to an inaccuracy or omission in the documentation, the procedure given for performing a reset of the chipset did not work. This meant that the only was to ensure the card was in a known state was to perform a full, cold boot of the entire machine. In addition, the documentation did not make it clear the exact offsets of the chipset registers. A future group has already been planned who will hopefully be able to work with CLARiiON and Emulex engineers to overcome these problems.

4.2 Linux Fibre Channel

At the beginning of the project, the Fibre Channel layer for Linux was in it's beginning stages. Since then, several core Linux developers have begun work on a layer which, conceptually, is very close to the layer we envisioned. Future groups working on this project should most likely write their driver to this layer, rather than to an individual subsystem (such as the SCSI layer, as we did) or attempt to implement this layer themselves.

4.3 Stability and Performance Testing

Unfortunately, due to several setbacks, we were unable to produce a driver capable of giving any meaningful test results. However, future groups should plan to perform and document as much stability and performance testing as possible.

4.4 Development Environment

For coding, all members of the group opted to use GNU Emacs running on Linux machines. GNU Emacs provides a very nice environment for C coding, including such features as keyword highlighting, automatic indentation, and automatic interaction with the CVS and RCS source control systems.

For source control, CVS was used. With a small groups such as ours, CVS combined with plenty of communication to avoid conflicts proved efficient for storing all work, including the code, project proposal, and writeup. The CVS repository was stored on a separate server which also served as a boot server for the development machine. The AViiON server was configured to boot the Linux kernel from a floppy and then load the rest of the system via NFS-Root to the CVS server. The local disk space on the AViiON server was used only for scratch space, temporary files, and building. This freed us from the worry of having to reload the operating system on the development machine when we crashed it during development. It also provided a stable place to keep our source repository and web site.

Chapter 5

The Emulex LP6000 Driver

The Emulex Light Pulse 6000 (LP6000) card is a high performance PCI Fibre Channel host adapter. It supports all Fibre Channel topologies (point to point, arbitrated loop, switch fabric), multiple Upper Layer Protocols (ULP), including both network (such as IP) and storage (such as FCP) loaded simultaneously. It has a maximum transfer rate of 1.0625 Gb/s at full duplex over either fiber optics or copper serial wire. The card is driven by the FireFly chipset.

Programming the card is done by means of the Service Level Interface Memory (SLIM). This is a region of memory on the LP6000 card that is mapped into the host main memory. The SLIM is further divided into 3 main parts: the mailbox, the I/O Control Block (IOCB) rings, and the registers. The registers are used for checking time critical status portions of the card, such as the interrupt status, and some commands to the card,

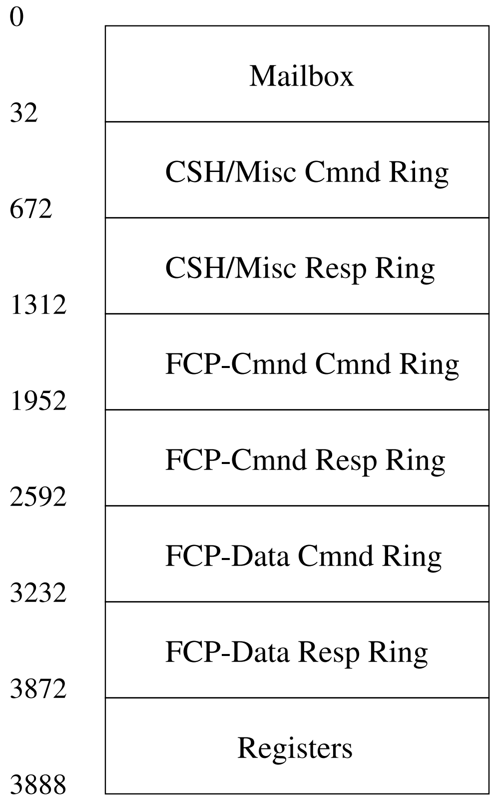

Figure 5.1: SLIM Partitioning Scheme

such as resetting the card. The mailbox is used for sending and receiving more complex commands, such as partitioning the SLIM and reading the NVRAM parameters. The IOCB rings are used for sending and receiving ULP packets.

5.1 Registers

The FireFly chipset has four 32 bit registers in the SLIM. The registers are used for configuring, controlling, and checking the status of the card.

The Host Attention Register (HAR) is used by the FireFly chipset to notify the host of a condition that requires attention. The sources can be any one of the IOCB rings, the mailbox, link attention, or an error.

The Chip Attention Register (CAR) is used by the host to notify the FireFly chipset of a condition that requires it's attention, typically of a command that requires processing. The source can be any one of the IOCB rings, or the mailbox.

The Control Register is used to initialize the chipset, and indicate which events should generate interrupts. The potential interrupts sources can be the mailbox, any of the IOCB rings, link attention, or errors. Each of these sources can be configured individually to generate interrupts or not. If interrupts are not enabled for any sources, then the host is expected to poll the status of that source periodically.

The Status Register is used by the FireFly chipset to indicate the status of the chipset. The conditions reported include the readiness of the chipset, readiness of the mailbox, and the various error conditions.

5.2 The Mailbox

The mailbox is used for sending commands to the FireFly chipset. The mailbox consists of eight 32 bit words mapped into host memory as part of the SLIM. It is primarily used during the initialization sequence, including partitioning the IOCB rings within the SLIM, and performing the fabric login.

The mailbox contains several fields. The first 32 bit word contains the return code, the command code, and the host contention (HC) and OWN bits. The HC bit is used as a mutex to exclude other processes on the host machine from accessing the host adapter when it is in use. The OWN bit is set to signal to the FireFly chipset that it may read the command and execute it. The chipset will clear the OWN bit when the command is complete. The remaining part of the mailbox contains any parameters necessary for the command being executed, as well as any result values.

5.3 I/O Control Block Rings

The IOCB rings are used for the actual transfer of data between the host and the card. The SLIM can contain up to 4 IOCB rings. Each ring is configured to receive a specific type of Fibre Channel message. Since our driver is oriented towards FCP only, we used 3 sets of rings: one for Fibre Channel command packets, one for Fibre Channel data packets, and one for CSH commands and any other types. Each set contains two individual rings,

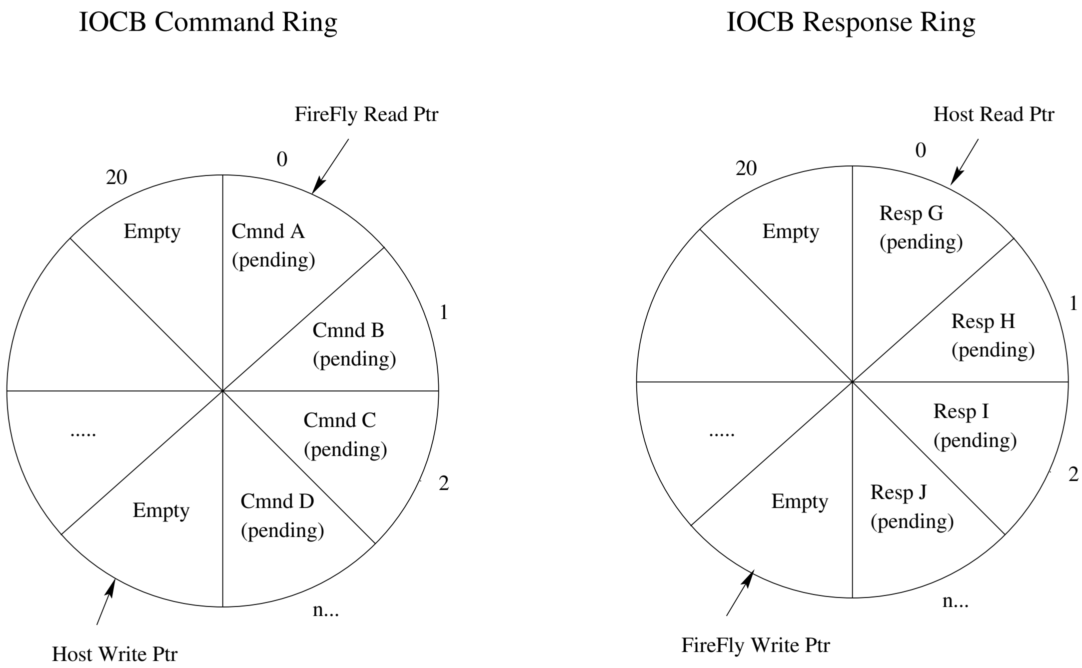

Figure 5.2: IOCB Ring Set

one for sending IOCB's, and one for receiving response IOCB's.

Each ring is logically presented as a circular buffer. To send a command, the driver must wait for the current entry in the buffer is clear. This can be done by enabling interrupts, which means that the host will receive an interrupt whenever a ring entry is freed, or by polling the OWN bit on the IOCB entry. The IOCB is then written into the ring, and the OWN bit is set, indicating that the transfer is complete and the chipset may start reading it.

A similar procedure is used for receiving IOCB response entries. If IOCB response interrupts have been enabled, then the host will receive an interrupt when a new response is available, otherwise the host must poll the OWN bit on the current entry. Once it is determined that the response entry can be read, the entire entry should be read from the SLIM into host memory, and then the OWN bit should be set, indicating to the chipset that it may re-use the entry.

Each IOCB entry consists of 8 32 bit words. Words 6 and 7 contain the various status bits, including a unique tag used to associate the IOCB command and it's corresponding response. The host is responsible for ensuring that each tag is unique. Words 0 through 6 are command dependent. Most typically, for data transfers, it will contain one or more Buffer Descriptor Entries (BDE). The BDE's are used to point to the specific region of memory that the card is to transfer data to or from.

Chapter 6

Conclusions and Future Work

Our primary goal for this project was simply to get a working, stable device driver for the LP6000 FC card. However, with a card as advanced and functional as this one, and with all of the features of Fibre Channel, there exists sufficient material for several future projects.

6.1 Current Driver Status

While a great deal of work has been accomplished with the LP6000 card, there is still more work that needs to be done to achieve a fully functional driver. Most of the problems we had were due to not so much an actual lack of documentation, but the fact that the documentation itself is either ambiguous or simply lacking altogether in a few key sections.

The single greatest problem was resetting the FireFly chipset. While the documentation does detail a procedure for resetting the card, it simply did not work for our card. Every time the driver attempted to perform a reset, the card never returned to the ready state. There are two probably causes for this. First, is simply that the documentation was inaccurate in how to reset the card, either leaving out some critical step, or a change that was made since the documentation was released.

Another possibility, however, is that we were not actually writing to the registers we thought we were. While the documentation specified where the registers began, it did not say in which order the registers appeared, or even if they were contiguous in the given space. We therefore had to dump the entire region out, and take a guess at which area was which register. We therefore cannot say for certain whether the driver is writing to the correct register, or simply at some area with completely undefined results.

We got around the reset problem by simply forcing a cold reboot of the machine. While this significantly slowed down development, it allowed us to proceed. Once we began doing this, we were able to get mailbox commands working. From here, the driver was able to get a fair amount of information off of the card, including link status, and the NVRAM parameters.

The last significant low-level section that needs work is the IOCB commands. While we believe the code in place is mostly correct, it still has a few bugs left in, and we believe there are also problems due to our lack of knowledge of the registers. Once this is completed, all low-level facilities to talk to the card will be in place, leaving only higher level code to be written.

Depending on the status of the Linux Fibre Channel layer, most of these facilities may even be available already.

6.2 Ideal Linux Fibre Channel Subsystem

For this project, the goal was simply to write a device driver to allow the card to drive a high end RAID. In practice, this meant that we were writing a SCSI driver to talk through the Fibre Channel layer. However, high-end Fibre Channel cards such as the LP6000 can easily support multiple Upper Layer Protocols (ULP) active at the same time, one or more per IOCB ring. Some of the more commonly used ULP's are FCP (SCSI over Fibre Channel), IP, and 802.3. A full-featured FC subsystem for Linux should include support for a single ULP module loaded for multiple FC cards, and multiple ULP's loaded on a single FC card (see Figure 6.1). Each card specific driver should advertise ULP specific functionality implemented in hardware. Each ULP should be able to use any hardware accelerated functions, while also providing alternative routines that use only generic FC functions.

Since we began this project, other Linux kernel developers have begun work on a Linux Fibre Channel subsystem, which appears to be developing very similarly to the model which we had envisioned.

At the bottom level (3) are the card specific drivers. These drivers perform any initialization, provide a way to pass the card FC commands, and advertise any functions which take advantage of hardware acceleration. They

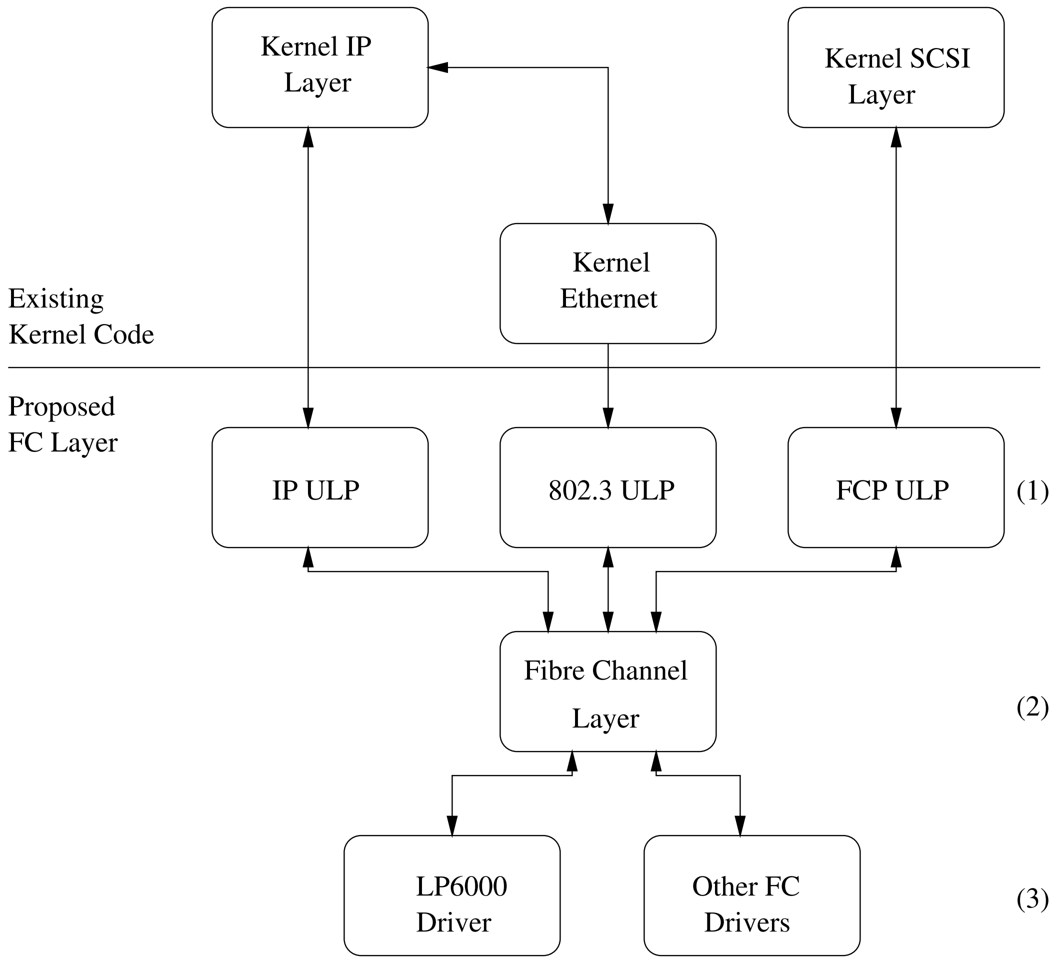

Figure 6.1: Ideal Linux FC Layer

must register all this information, plus any other relevant facts, such as what topology is being used, with the FC mid-level.

The FC mid-level (2), in our initial design, is very thin. It may in fact be so thin that as to warrant being removed altogether in the final version. Essentially, it simply contains information to allow the ULP's and low-level drivers to communicate with each other. It may also contain features common to all cards, such as a card-global semaphore and task queue lists.

The upper portion of the FC code contains the various ULP's (1). They contain the protocol specific logic - SCSI for the FCP module, IP for the IP module, etc. Each ULP then presents itself to an existing kernel subsystem as a low level driver for that subsystem. The FCP driver pretends it's a low level SCSI driver, the IP ULP presents itself as a network card, etc. For any functions which may be time critical, each module should also contain an implementation of each function that uses only generic FC commands. These functions should be registered in a function table with the FC mid-level, and then only called from the function table presented by the FC mid-level. This will allow the FC mid-level to override the generic functions provided by the ULP with functions provided by the low level drivers that use hardware accelerated card specific functions.

6.3 Stability

One of the most critical features of any server, whether high or low end, is stability. Without a proven high level of stability, the users cannot depend on the server to do their work, and the organization depending on the server ends up losing valuable time, and often money.

Linux is an excellent overall choice for stability. On quality hardware, Linux has high recorded uptimes—in a few cases exceeding 400 days. In order for our driver to be considered a success, we must ensure that we meet that same high level of reliability.

Despite this, Linux is not a fault tolerant, high availability OS, and many conditions, particularly hardware errors, will cause Linux to fail. Fibre Channel makes application fail-over much easier than SCSI, since a topology that includes multiple servers, disk arrays, and redundant paths between them is relatively easy to construct. For future work, application fail-over, particularly for databases, would be an ideal use of this technology.

6.4 Performance

Fibre Channel disk arrays are high performance devices, designed to output data at extremely high rates. To fully use the bandwidth these devices are capable of, one must often push the server hardware near its theoretical limits. One must therefore make sure that the code is as streamlined and efficient as possible, while still ensuring that it does not sacrifice stability under any circumstances.

In addition to using established I/O benchmark utilities, one should also employ robust database benchmarking. Databases are very I/O intensive, particularly online transaction processing (OLTP) databases. They tend to stress the logs more than anything else.

Using raw disks has traditionally been a way to deal with this requirement. There is a small speed gain, due to not having to go through the filesystem layer in the kernel. It has fallen out of favor with DBA's and vendors for several reasons. First, administration becomes harder. Second, the performance gain is typically 10-15%, while a two disk stripe would yield a throughput increase of 100%. Even without methods such as striping, the price to performance ratio in modern hardware is such that the small gain has a very low cost difference in server class hardware. Lastly, disk array vendors have already realized the problem and have implemented code that speeds disk writes by concatenating them together into the log. By using large disk arrays, combined with logical volume managers, it becomes possible to linearly scale I/O performance without making management of the system infeasible.

Databases and multimedia streaming currently represent the major market for disk arrays. In light of the recent decision of the major relational database vendors to support Linux, there will probably be demand for research in that area.

Bibliography

[1] CLARiiON Advanced Storage Solutions. http://www.clariion.com/

[2] Linux Homepage. http://www.linux.org/

[3] Free Software Foundation. http://www.fsf.org/

[4] GNU General Public License. http://www.gnu.org/copyleft/gpl.html

[5] Rubini, Alessandro. Linux Device Drivers. Sebastopol, CA: O'Reilly & Associates, Inc., 1998.

[6] Morano, Michael and Savela, Russ. I/O Subsystems Technical Report. http://angus.ind.wpi.edu/fp/docs/io-report/io-report/index.html, Worcester, MA: Worcester Polytechnic Institute, 1998.

[7] Patterson, Gibson and Katz, Randy. RAID: High-Performance, Reliable Secondary Storage. http://sunsite.berkeley.edu/Dienst/Repository/2.0/Body/ncstrl.ucb/CSD-93-778/postscript. University of California Berkeley, 1993.

[8] Coveney, Christopher and Kaufmann, Aviv. RAID System Queuing Model. Worcester, MA: Worcester Polytechnic Institute, MQP 98D415M, 1998.

[9] Fibre Channel Linux Driver. http://www.iol.unh.edu/consortiums/fc/fc_linux.html

[10] Fibre Channel Association. http://www.fibrechannel.com/

[11] Linux Kernel Hacker's Guide. http://www.redhat.com:8080/